1. **Как создать объект класса String, какие конструкторы класса String вы знаете?**

Строка представляет собой последовательность символов.  Физически объект String представляет собой ссылку на область в памяти, в которой размещены символы.

 String str1 = "Java";

    String str2 = new String(); // пустая строка. Можно **создать** пустой **объект класса String.**

    String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});

    String str4 = new String(new char[]{'w', 'e', 'l', 'c', 'o', 'm', 'e'}, 3, 4);//3 -начальный индекс, 4 -кол-во символов

    System.out.println(str1); // Java

    System.out.println(str2); //

    System.out.println(str3); // hello

    System.out.println(str4); // come

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String--)()

Инициализирует вновь созданный Stringобъект так, чтобы он представлял пустую последовательность символов.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-byte:A-)(byte[] bytes)

Создает новый String, декодируя указанный массив байтов с использованием кодировки платформы по умолчанию.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-byte:A-java.nio.charset.Charset-)(byte[] bytes, **[Charset](https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html" \o "class in java.nio.charset)** charset)

Создает новый String, декодируя указанный массив байтов с использованием указанной [**кодировки**](https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html).

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-byte:A-int-int-)(byte[] bytes, int offset, int length)

Создает новый String, декодируя указанный подмассив байтов с использованием кодировки платформы по умолчанию.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-byte:A-int-int-java.nio.charset.Charset-)(byte[] bytes, int offset, int length, **[Charset](https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html" \o "class in java.nio.charset)** charset)

Создает новый String, декодируя указанный подмассив байтов с использованием указанной [**кодировки**](https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html).

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-byte:A-int-int-java.lang.String-)(byte[] bytes, int offset, int length, [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) charsetName)

Создает новый String, декодируя указанный подмассив байтов с использованием указанной кодировки.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-char:A-)(char[] value)

Выделяет новый объект Stringтак, чтобы он представлял последовательность символов, которая в данный момент содержится в аргументе массива символов.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-char:A-int-int-)(char[] value, int offset, int count)

Выделяет новый объект String, содержащий символы из подмассива аргумента массива символов.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-int:A-int-int-)(int[] codePoints, int offset, int count)

Выделяет новый объект String, содержащий символы из подмассива аргумента массива [**кодовых точек Unicode**](https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#unicode) .

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-java.lang.String-)(**[String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html" \o "class in java.lang)** original)

Инициализирует вновь созданный Stringобъект, чтобы он представлял ту же последовательность символов, что и аргумент; другими словами, вновь созданная строка является копией строки аргумента.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-java.lang.StringBuffer-)(**[StringBuffer](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html" \o "class in java.lang)** buffer)

Выделяет новую строку, содержащую последовательность символов, которая в данный момент содержится в аргументе строкового буфера.

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String-java.lang.StringBuilder-)(**[StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html" \o "class in java.lang)** builder)

Выделяет новую строку, содержащую последовательность символов, которая в данный момент содержится в аргументе построителя строк.

**1.1. Что такое строковый литерал?**

**Строковые литералы** — это набор символов, заключенных в двойные кавычки.

Все литералы — это примитивные значения (строки, числа, символы, булевы значения). Нельзя создать литерал-объект. Единственный литерал, связанный с объектом — это null. В соответствии с примитивами, литералы также делятся на разделы:

1. Числовые:
   1. Целочисленные;
   2. С плавающей точкой;
2. Строковые;
3. Символьные;
4. Логические.

**1.2. Объясните, что значит “упрощенное создание объекта String”?**

Существует четыре различных способа создания объектов в java:

**Один. Использование ключевого слова  new.**   
Это самый распространенный способ создания объекта в java. Таким образом создается почти 99% объектов.

MyObject object = new MyObject();

**Б. Использование Class.forName()**Если мы знаем имя класса &, если он имеет открытый конструктор по умолчанию, мы можем создать объект таким образом.

MyObject object = (MyObject) Class.forName("subin.rnd.MyObject").newInstance();

**C . Использование clone()**clone() можно использовать для создания копии существующего объекта.

MyObject anotherObject = new MyObject();

MyObject object = (MyObject) anotherObject.clone();

**Д.** **Использование object deserialization**  
Десериализация объекта - это не что иное, как создание объекта из его сериализованной формы.

ObjectInputStream inStream = new ObjectInputStream(anInputStream );

MyObject object = (MyObject) inStream.readObject();

1. **Можно ли изменить состояние объекта типа String?**

При работе со строками важно понимать, что объект String является неизменяемым (immutable). То есть при любых операциях над строкой, которые изменяют эту строку, фактически будет создаваться новая строка.

Для переменных примитивного типа это означает, что однажды присвоенное значение не может быть изменено. Для ссылочных переменных это означает, что после присвоения объекта, нельзя изменить ссылку на данный объект. **Это важно! Ссылку изменить нельзя, но состояние объекта изменять можно.**

**2.1.Что происходит при попытке изменения состояния объекта типа String?**

**2.2.Можно ли наследоваться от класса String?**

public final class String - это final класс, которые не наследуются.

**2.3. Как вы думаете, почему строковые объекты immutable?**

Это неизменяемый (immutable) и финализированный тип данных в Java и все объекты класса String виртуальная машина хранит в пуле строк.

**1) Безопасность.**

String широко используется , как параметр для многих классов Java, в частности для открытия сетевых соединений, подключений к БД, открытию файлов и пр. И если бы строка изменялась, то мы могли получить доступ к объекту (файлу например), на который мы имеем право, затем изменить строку с именем (случайно или намеренно) и получить доступ уже к другому файлу.

Так же String используется в механизме загрузки файлов, и это - фундаментальный аспект. И если бы строка изменялась, то запрос на загрузку "java.io.Writer" мог бы быть изменён на "DiskErasingWriter".

**2) Hashcode**

Из-за того, что строка не изменяется, она кэширует свой хэшкод и не вычисляет его каждый раз, когда мы его вызываем, что делает строку очень быстрой как ключ для hashmap.

**3) Многопоточность**

immutable делает экземпляры строк thread-safe.

1. **Объясните, что такое кодировка?**

У каждого символа есть код (число). Именно поэтому тип char считается не только символьным, но и числовым типом.

Т.е. каждой букве или каждому символу соответствует какое-то число.

Символ можно преобразовать в число, а число в символ. Java вообще практически не видит разницы между ними:

**char** c = 'A'; // код(число) буквы А – 65

c++; // теперь с содержит число 66 – код буквы B

**3.1. Какие кодировки вы знаете?**

UTF8

Unicod - универсальная

UTF16

Windows -1251, КОИ8-Р

**3.2. Как создать строки в различной кодировке?**

**Метод - byte**[] getBytes()

String s = "Good news everyone!";

**byte**[] buffer = s.**getBytes**()

**Метод - byte**[] getBytes(**Charset** **charset**)

String s = "Good news everyone!";

Charset koi8 = Charset.forName("KOI8-R");

**byte**[] buffer = s.**getBytes**(koi8);

**Метод - byte**[] getBytes(**String** **charsetName**)

String s = "Good news everyone!";

**byte**[] buffer = s.**getBytes**("Windows-1251")

1. **Что такое пул литералов?**

Пул строк (**String Pool**) — это множество строк в кучи (**[Java Heap Memory](https://javadevblog.com/chto-takoe-heap-i-stack-pamyat-v-java.html" \t "_blank)**). Мы знаем, что String — особый класс в java, с помощью которого мы можем создавать строковые объекты.

Когда мы используем двойные кавычки, чтобы создать новую строку, то первым делом идет поиск строки с таким же значением в пуле строк. Если java такую строку нашла, то возвращает ссылку, в противном случае создается новая строка в пуле, а затем возвращается ссылка.

Однако использование оператора new заставляет класс String создать новый объект String. После этого можем использовать метод intern(), чтобы поместить этот объект в пул строк или обратиться к другому объекту из пула строк, который имеет такое же значение.

**4.1. Как строки заносятся в пул литералов?**

Когда мы используем двойные кавычки для создания строки, сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка. Тем не менее, когда мы используем оператор new, мы принуждаем класс String создать новый объект строки, а затем мы можем использовать метод intern() для того, чтобы поместить строку в пул, или получить из пула ссылку на другой объектString с таким же значением.

**4.2. Как занести строку в пул литералов и как получить ссылку на строку, хранящуюся в пуле литералов?**

**4.3. Где хранится (в каком типе памяти) пул литералов в Java 1.6 и Java 1.7?**

**5. В чем отличие объектов классов StringBuilder и StringBuffer от объектов класса String?**

В отличие от String, объекты типа StringBuffer и StringBuilder могут быть изменены снова и снова, оставляя после себя множество новых неиспользуемых объектов.

**5.1. Какой из этих классов потокобезопасный?**

  Выгода от применения **StringBuilder** связана с более высокой производительностью. Однако в случае разработки многопоточных программ вы должны использовать **StringBuffer**, а не **StringBuilder**.

**Операции с StringBuffer потокобезопасны и синхронизированы**, а методы StringBuilder не потокобезопасны. Поэтому когда несколько нитей работают с одной строкой, мы должны использовать StringBuffer, но в однопоточном окужении мы должны использовать StringBuilder.

Рекомендуется использовать StringBuilder всякий раз, когда это возможно, потому что он быстрее, чем StringBuffer в Java. Однако, если необходима безопасность потоков, наилучшим вариантом являются объекты StringBuffer.

**5.2. Как необходимо сравнивать на равенство объекты классов StringBuilder и StringBuffer и почему?**

Классы **StringBuilder** и **StringBuffer** являются “близнецами” и по своему предназначению близки к классу **String**, но, в отличие от последнего, содержимое и размеры объектов классов**StringBuilder** и  **StringBuffer** можно изменять.

Основным и единственным отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего. В версии 1.5.0 был добавлен непотокобезопасный (следовательно, более быстрый в обработке) класс **StringBuilder**, который следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

С помощью соответствующих методов и конструкторов объекты классов **StringBuffer**,**StringBuilder** и **String** можно преобразовывать друг в друга. Конструктор класса **StringBuffer** (также как и **StringBuilder**) может принимать в качестве параметра объект **String** или неотрицательный размер буфера. Объекты этого класса можно преобразовать в объект класса **String** методом **toString()**или с помощью конструктора класса **String**.

При создании объекта **StringBuffer** конструктор по умолчанию автоматически резервирует некоторый объем памяти (16 символов), что в дальней­шем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе. Если длина строки **StringBuffer**после изменения превышает его размер, то ёмкость объекта автоматически увеличивается, оставляя при этом резерв для дальнейших изменений. С помощью метода **reverse()** можно быстро изменить порядок символов в объекте.

Если метод, вызываемый объектом **StringBuffer**, производит измене­ния в его содержимом, то это не приводит к созданию нового объекта, как  
в случае объекта **String**, а изменяет текущий объект **StringBuffer**.

Для класса **StringBuffer** не переопределены методы **equals()** и **hashCode()**, т.е. сравнить содержимое двух объектов невозможно, к тому же хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**.

Рекомендуется использовать StringBuilder всякий раз, когда это возможно, потому что он быстрее, чем StringBuffer в Java. Однако, если необходима безопасность потоков, наилучшим вариантом являются объекты StringBuffer**.**

**6. Что такое Unicode?**

**Unicode** — это международный стандарт кодировки символов, который может представлять большинство письменных языков по всему миру. Юникод использует шестнадцатеричное для представления символа. Unicode — это 16-битная система кодирования символов. Наименьшее значение равно \ u0000, а самое высокое значение равно \ uFFFF.

**7. Какие методы класса String используются для работы с кодовыми точками? Как вы думаете, когда следует их использовать?**

**Regular Expressions**

1. **Расскажите, что представляет собой регулярное выражение?**

Регулярные выражения – это специальные последовательности символов, которые помогают нам находить строки или множества строк, используя специальные синтаксические шаблоны.

Регулярные выражения в Java обеспечиваются пакетом **java.util.regex**.

В пакете **java.util.regex**ключевыми являются 3 класса:

* **Matcher**Этот класс интерпретирует шаблон и предоставляет определяет совпадения в водимой строке.
* **Pattern**Класс Pattern предоставляет нам скомпилированный вариант регулярного выражения.
* **PatternSyntaxException**Этот класс предоставляет нам непроверяемые исключения, которые указывают нам на синтаксическую ошибку в нашем регулярном выражении.

**1.1.Что такое метасимволы регулярного выражения?**

Основой синтаксиса регулярных выражений является тот факт, что некоторые символы встречающиеся в строке рассматриваются не как обычные символы, а как имеющие специальное значение (т.н. метасимволы). Именно это решение позволяет работать всему механизму регулярных выражений. Каждый метасимвол имеет свою собственную роль.

**1.2. Какие вы знаете классы символов регулярных выражений?**

Пакет java.util.regex исходно состоит из следующих трех классов:

* **Pattern Class** – объект класса Pattern представляет скомпилированное представление регулярного выражения. В классе Pattern публичный конструктор не предусмотрен. Для создания шаблона, вам сперва необходимо вызвать один из представленных публичных статичных методов **compile()**, который далее произведет возврат объекта класса Pattern. Регулярное выражение в данных методах принимается как первый аргумент.
* **Matcher Class** – объект класса Matcher представляет механизм, который интерпретирует шаблон, а также производит операции сопоставления с вводимой строкой. Аналогично классу Pattern, Matcher не содержит публичных конструкторов. Объект класса Matcher может быть получен путем вызова метода **matcher()** на объекте класса Pattern.
* **PatternSyntaxException** – объект класса PatternSyntaxException представляет непроверяемое исключение, которое обозначает синтаксическую ошибку в шаблоне регулярного выражения.

**1.3. Что такое квантификаторы?**

Метасимволы для обозначения количества символов – квантификаторы. Квантификатор всегда следует после символа или группы символов.

**1.4. Какие логические операторы регулярных выражений вы знаете?**

| **Выражение** | **... соответствует** |
| --- | --- |
| XY | X, за которым идёт Y |
| X|Y | Либо X, либо Y |
| (XY) | XY как отдельная группа |

**1.5. Что значит “якорь” для регулярного выражения?**

# Якоря: начало строки ^ и конец $

У символов каретки ^ и доллара $ есть специальные значения в регулярных выражениях. Они называются «якоря» (anchors).

Каретка ^ означает совпадение с началом текста, а доллар $ – с концом.

К примеру, давайте проверим начинается ли текст с Mary:

let str1 = "Mary had a little lamb";

alert( /^Mary/.test(str1) ); // true

Шаблон ^Mary означает: «начало строки, затем Mary».

Аналогично можно проверить, кончается ли строка словом snow при помощи snow$:

let str1 = "it's fleece was white as snow";

alert( /snow$/.test(str1) ); // true

В конкретно этих случаях мы могли бы использовать и методы строк startsWith/endsWith. Регулярные выражения следует применять, когда нужна проверка сложнее.

## [Проверка на полное совпадение](https://learn.javascript.ru/regexp-anchors#proverka-na-polnoe-sovpadenie)

Оба якоря вместе ^...$ часто используются для проверки, совпадает ли строка с шаблоном полностью. Например, чтобы определить, в правильном ли формате пользователь ввёл данные.

Проверим, что строка является временем в формате 12:34, то есть две цифры, затем двоеточие, затем ещё две цифры.

На языке регулярных выражений это \d\d:\d\d:

let goodInput = "12:34";

let badInput = "12:345";

let regexp = /^\d\d:\d\d$/;

alert( regexp.test(goodInput) ); // true

alert( regexp.test(badInput) ); // false

Здесь совпадение с \d\d:\d\d ищется не где-то посередине текста, а сразу после начала строки ^, и после него должен быть сразу конец строки $.

То есть, вся строка – это как раз время в нужном нам формате.

Поведение якорей меняется, если присутствует флаг m. Мы рассмотрим этот флаг в следующей статье.

**У якорей нулевая ширина**

Якоря ^ и $ – это проверки. У них нулевая ширина.

Другими словами, они не добавляют к результату поиска символы, а только заставляют движок регулярных выражений проверять условие (начало/конец текста).

**2. Какие java-классы работают с регулярными выражениями?**

**Macther**

**Patern, PatternSuntaxException**

**2.1. В каком пакете они расположены?**

В пакете **java.util.regex**ключевыми являются 3 класса:

* **Matcher**Этот класс интерпретирует шаблон и предоставляет определяет совпадения в водимой строке.
* **Pattern**Класс Pattern предоставляет нам скомпилированный вариант регулярного выражения.
* **PatternSyntaxException**Этот класс предоставляет нам непроверяемые исключения, которые указывают нам на синтаксическую ошибку в нашем регулярном выражении.

**2.2. Приведите пример анализа текста с помощью регулярного выражения и поясните код примера.**

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main {

public static void main (String[] args) {

**Pattern pattern1 = Pattern.compile ("[x-z]+");//**Поиск будет происходить от x до z включительно.

//Поиск будет происходить только по символам нижнего регистра.

//Чтобы отключить чувствительность к регистру, можно использовать Pattern.CASE\_INSENSITIVE.

**Matcher matcher1 = pattern1.matcher ("x y z 1 2 3 4 ");**

System.out.println (matcher1.find()); //Поиск любого совпадения с шаблоном.

//Выводится значение true, так как в строке есть символы шаблона.

**Matcher matcher2 = pattern1.matcher ("X Y Z 1 2 3 4");**

System.out.println (matcher2.find()); //Выводится значение false.

//Так как в строке нет символов, подходящих по шаблону.

**Pattern pattern2 = Pattern.compile ("[a-zA-Z0-9]");**

//Добавляется поиск по символам нижнего и верхнего регистра, а также цифр.

**Matcher matcher3 = pattern2.matcher ("A B C D X Y Z a b c d x y z 1 2 3 4");**

System.out.println (matcher3.find()); //Выводится значение true

**3. Что такое группы в регулярных выражениях?**

*Группы фиксации* являются способом обработать многократные символы как единый блок. Они создаются, помещая символы, которые будут сгруппированы в ряде круглых скобок. Например, регулярное выражение (dog) создает единственную группу, содержащую буквы "d" "o" и "g". Часть строки ввода, которая соответствует группу фиксации, будет сохранена в памяти для более позднего отзыва через обратные ссылки.

Чтобы узнать, сколько групп присутствует в выражении, вызовите groupCount метод на объекте matcher.

**3.1. Как нумеруются группы?**

Группы сбора нумеруются посредством определения числа открывающих круглых скобок слева направо. Так, в выражении ((A)(B(C))) присутствуют четыре подобные группы:

* ((A)(B(C)))
* (A)
* (B(C))
* (C)

**3.2. Что представляет собой группа номер 0(ноль)? Приведите пример с использованием групп регулярного выражения.**Есть также специальная группа, группа 0, который всегда представляет все выражение. Эта группа не включается в общее количество, о котором сообщают groupCount. Начинающие группы (? чистые, *негруппы фиксации*, которые не получают текст и не рассчитывают к групповому общему количеству.